Examen de Programación Concurrente - Clave a

Septiembre 2009

Departamento de Lenguajes, Sistemas Informáticos e Ingeniería de Software

Normas

Este examen mezcla preguntas tipo test y preguntas de respuesta corta. Consta de **9 preguntas** en **4 páginas**. La puntuación total del examen es de **10 puntos**. La duración total es de **una hora y media**. El examen debe contestarse en las **mismas hojas**. No olvidéis rellenar **apellidos, nombre y número de matricula** en cada hoja.

Sólo hay una respuesta válida por pregunta tipo test. Toda pregunta en que se marque más de una respuesta se considerará incorrectamente contestada. Toda pregunta incorrectamente contestada restará del examen una cantidad de puntos igual a la puntuación de la pregunta dividido por el número de alternativas en la misma.

La solución al examen se proporcionará antes de la revisión. Las calificaciones se darán a conocer el **22 de septiembre**. La revisión del examen tendrá lugar el **24 de septiembre**.

Cuestionario

(1 punto) 1. Dado el siguiente **CTAD** (no se muestra el interfaz pero no contiene otras operaciones que las especificadas, el tercer componente del tipo es una tabla):

TIPO: $Barrera = (abierta : \mathbb{B} \times color\_abierto : Color \times quieren\_pasar : Color \nrightarrow \mathbb{Z})$

DONDE: $Color = \{rojo, verde, azul\}$
INVARIANTE: $\forall b \in Barrera.\ \forall c \in Color.\ 0 \leq b.quieren\_pasar(c) \wedge b.quieren\_pasar(c) \leq 10$

INICIAL(b): $\neg b.abierta \wedge \forall c \in Color.\ b.quieren\_pasar(c) = 0$
CPRE: $\neg b.abierta$
SolicitarPaso(b,c)
POST: $b^{sal}.abierta = b^{ent}.abierta \wedge b^{sal}.color\_abierto = b^{ent}.color\_abierto$
$\wedge\ b^{sal}.quieren\_pasar = b^{ent}.quieren\_pasar \oplus \{c \mapsto b^{ent}.quieren\_pasar(c) + 1\}$
CPRE: $b.abierta \wedge b.color\_abierto = c$
Pasar(b,c)
POST: $b^{sal}.abierta = (b^{sal}.quieren\_pasar(c) > 0) \wedge b^{sal}.color\_abierto = b^{ent}.color\_abierto$
$\wedge\ b^{sal}.quieren\_pasar = b^{ent}.quieren\_pasar \oplus \{c \mapsto b^{ent}.quieren\_pasar(c) - 1\}$
CPRE: $\neg b.abierta$
Abrir(b,c)
POST: $b^{sal}.abierta = (b^{ent}.quieren\_pasar(c) > 0) \wedge b^{sal}.color\_abierto = c$
$\wedge\ b^{sal}.quieren\_pasar = b^{ent}.quieren\_pasar$

Supóngase un programa concurrente en el que los procesos respetan los siguentes protocolos (o esquemas de llamada): $SolicitarPaso(b, c_1); Pasar(b, c_1)$ y $Abrir(b, c_2)$ donde c_1 y c_2 son colores.

Se pide señalar la respuesta correcta (asumir, obviamente, que el invariante se cumple antes de la invocación de cada operación).

(a) ☐ El programa puede violar el invariante del recurso compartido sólo en la operación $SolicitarPaso$.

(b) ☐ El programa puede violar el invariante del recurso compartido sólo en la operación $Pasar$.

(c) ☐ El programa puede violar el invariante del recurso compartido sólo en la operación $Abrir$.

(d) ☐ Ninguna de las otras respuestas es correcta.

(1 punto) 2. **Se pide** marcar la afirmación correcta sobre el paso de mensajes síncrono.

(a) ☐ El envío y la recepción de un mensaje se terminan de ejecutar al mismo tiempo.

(b) ☐ El envío de un mensaje se ejecuta antes que su recepción.

(c) ☐ Pueden perderse mensajes.

(d) ☐ Los mensajes pendientes de ser recibidos quedan necesariamente almacenados en un *buffer* asociado al canal.

(1 punto) 3. El recurso del multibuffer admite insertar o extraer un número arbitrario de datos de un buffer acotado. Las condiciones de sincronización de ambas operaciones son que haya suficientes huecos para almacenar los datos que se quiere insertar y que haya suficientes elementos para retirar los datos que se quiere extraer. Se ha implementado el recurso mediante *rendez-vous* y paso de mensajes siguiendo la metodología enseñada en clase quedando el siguiente pseudocódigo de desbloqueo tras la *select* del servidor que encapsula el buffer:

```
      ...
   end select;
   << Bucle de desbloqueo de todos los procesos bloqueados para extraer >>;
   << Bucle de desbloqueo de todos los procesos bloqueados para insertar >>;
end loop;
```

Se pide: marcar la afirmación correcta.

(a) ☐ El esquema puede provocar que queden procesos bloqueados a pesar de que se cumple la CPRE.

(b) ☐ Hay que desbloquear alternativamente a procesos de diferente tipo: consumidor, productor, consumidor, productor, ...

(c) ☐ Es necesario desbloquear primero a los procesos bloqueados para insertar.

(d) ☐ El esquema de código es perfecto.

(1 punto) 4. Dado el siguiente programa concurrente

```
procedure Examen_2009sep is

   task type T (Y : Integer);

   T_1 : T (1);
   T_2 : T (2);
   X : Integer := 100;
```

```
   task body T is
      Z : Integer := Y;
   begin
      Z := Z + Y; X := X + Z;
   end T;

begin  --  Programa principal
   null;
end Examen_2009sep;
```

Se pide marcar la afirmación correcta.

(a) ☐ Es necesario asegurar exclusión mutua en el acceso a la variable `X`.

(b) ☐ Es necesario asegurar exclusión mutua en el acceso al parámetro `Y`.

(c) ☐ Es necesario asegurar exclusión mutua en el acceso a la variable `Z`.

(d) ☐ Ninguna de las otras respuestas es correcta.

(1 punto) 5. Se quiere implementar una operación `Op` de un recurso compartido, cuyas condiciones de concurrencia dependen de los parámetros de entrada, con el siguiente esquema (supóngase que el vector `V` ha sido convenientemente dimensionado e inicializado):

```
entry Op (X: T)
  when True is
  I: Natural := 0;
begin
  while V(I).Ocup loop
    I := I + 1;
  end loop;
  V(I).Ocup := True;
  V(I).Args := X;
  requeue Op_Ap(I);
end P;
```

```
entry Op_Ap
  (for J in Tipo_PID)
  (X: T)
  when CPRE (V(J).Args) is
begin
  V(J).Ocup := False;
  <<Sentencias para
    alcanzar la POST>>
end P_Ap;
```

Se pide marcar la afirmación correcta.

(a) ☐ Es una implementación correcta (asumiendo que las condiciones están bien implementadas).

(b) ☐ Se está ejecutando `V(J).Ocup := False` demasiado pronto: hay que esperar a que se acabe de ejecutar completamente la **`entry`** correspondiente para evitar que otro proceso se *apropie* del componente `J`-ésimo y "machaque" su contenido.

(1 punto) 6. El siguiente tipo de tareas `P` implementa un protocolo de acceso a una sección crítica.

```
task type P (I : PID);

task body P is
begin
   loop
      S;
      while Turno /= I loop null; end loop;
      Seccion_Critica;
      Turno := Turno + 1;
   end loop;
end P;
```

Debe asumirse que la sentencia `Turno := Turno + 1` es atómica y equivalente a
if `Turno = MAX_TASKS` **then** `Turno := 1;` **else** `Turno := Turno + 1;` **end if**.

Supóngase un programa concurrente con tantas tareas de tipo `P` como elementos tenga el tipo `PID`, todas ellas compartiendo compartiendo una variable `Turno` inicializada a `PID'First` y cada una de ellas con un índice `I` distinto.

Se pide marcar la afirmación correcta.

(a) ☐ Asumiendo la terminación de `S` y `Seccion_Critica`, el programa **no cumple** la propiedad de exclusión mutua en `Seccion_Critica`.

(b) ☐ Asumiendo la terminación de `S` y `Seccion_Critica`, el programa **no cumple** la propiedad de ausencia de interbloqueo.

(c) ☐ Asumiendo la terminación de `S` y `Seccion_Critica`, el programa **cumple** la propiedad de ausencia de inanición.

(d) ☐ Ninguna de las otras respuestas es correcta.

(1 punto) 7. A continuación se muestra un diseño de un sistema concurrente con una especificación formal de un recurso compartido (no se muestra el interfaz pero no contiene otras operaciones que las especificadas siendo el segundo argumento de las operaciones de tipo $\mathbb{Z}$) y tres tareas `T1`, `T2` y `T3` que comparten dicho recurso (variable `N : Notificacion`) y que, juntas, forman un programa concurrente.

TIPO: $Notificacion = \mathbb{Z}$
INICIAL(n)**:** $n = 0$

CPRE: Cierto
Notificar(n,x)
POST: $n^{sal} = x$

CPRE: $n \neq x$
Sincronizar(n,x)
POST: $n^{sal} = n^{ent}$

```
task body T1 is
begin
   Notificar (N, 1);
end T1;

task body T2 is
begin
   Sincronizar (N, 0);
   Put (Integer'Image (0));
end T2;

task body T3 is
begin
   Sincronizar (N, 1);
   Put (Integer'Image (1));
end T3;
```

Se pide marcar la afirmación correcta suponiendo que las operaciones `Put` son atómicas.

(a) ☐ El programa puede no terminar.

(b) ☐ "`01`" no es una salida posible del programa.

(c) ☐ "`0`" no es una salida posible del programa.

(d) ☐ "`10`" no es una salida posible del programa.

(1 punto) 8. Supóngase que se declara una variable global sin encapsularla dentro de un objeto protegido pero que el acceso a la misma desde varias tareas se realiza a través de entradas de un único objeto protegido.

Bajo las condiciones descritas, ¿está garantizada la exclusión mutua en el acceso a dicha variable?

¿Por qué no es una solución ideal al problema de garantizar exclusión mutua en el acceso a dicha variable?

(2 puntos) 9. Cuando varios procesos se comunican utilizando recursos compartidos aparecen dos tipos de sincronización: *exclusión mutua* y *sincronización condicional*. Describe en qué consiste cada uno de esos tipos de sincronización.